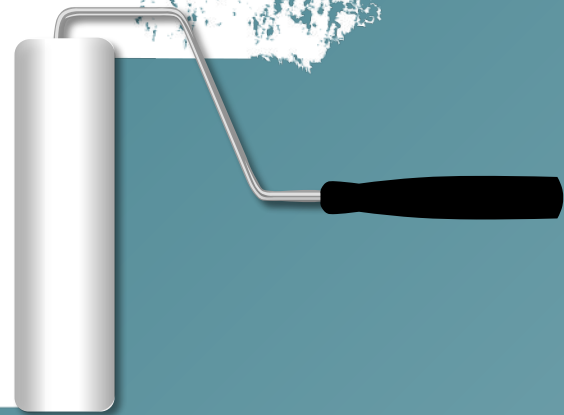


Cheap and Mostly Right



Introducing Probabilistic Metrics for Efficient Statistics on Large Streams



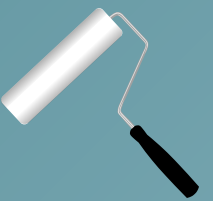
Why are we here?

- Put probabilistic metrics on your 'tech radar'
- Show how we used HyperLogLog to improve key metrics by 99%
- Introduce the 'family' of different probabilistic metrics

The TL;DR

These metrics all do one magic trick:

- Compute 'set/stream' metrics, e.g. # of unique values, freq. distribution, etc.
- *Without storing the whole set!*
- Instead, they store tiny, statistical *sketches*
- Trade accuracy for storage space / speed
- Aggregatable like sum/max/min/etc.
- Cheap, and mostly right!

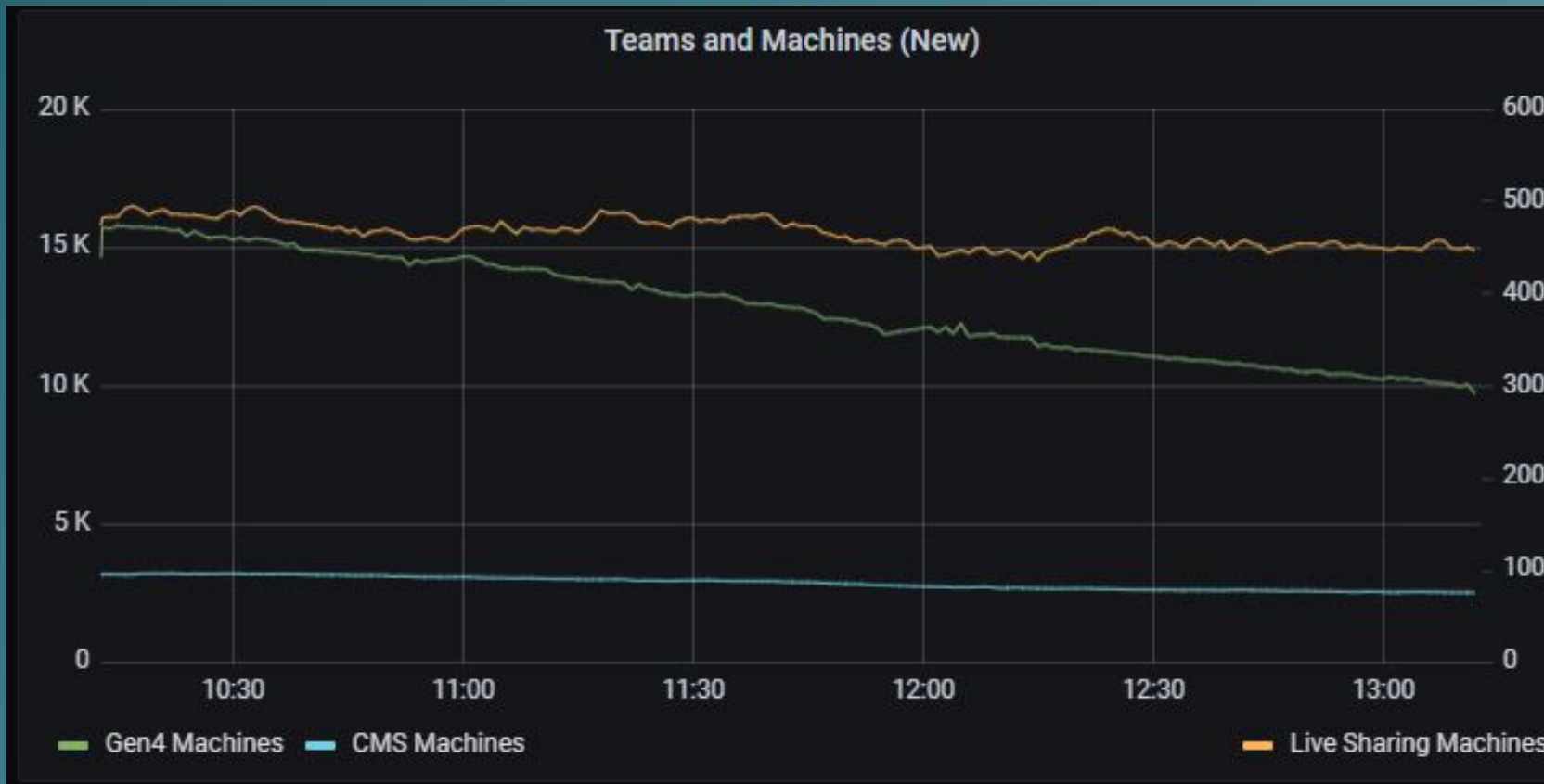




Our problem

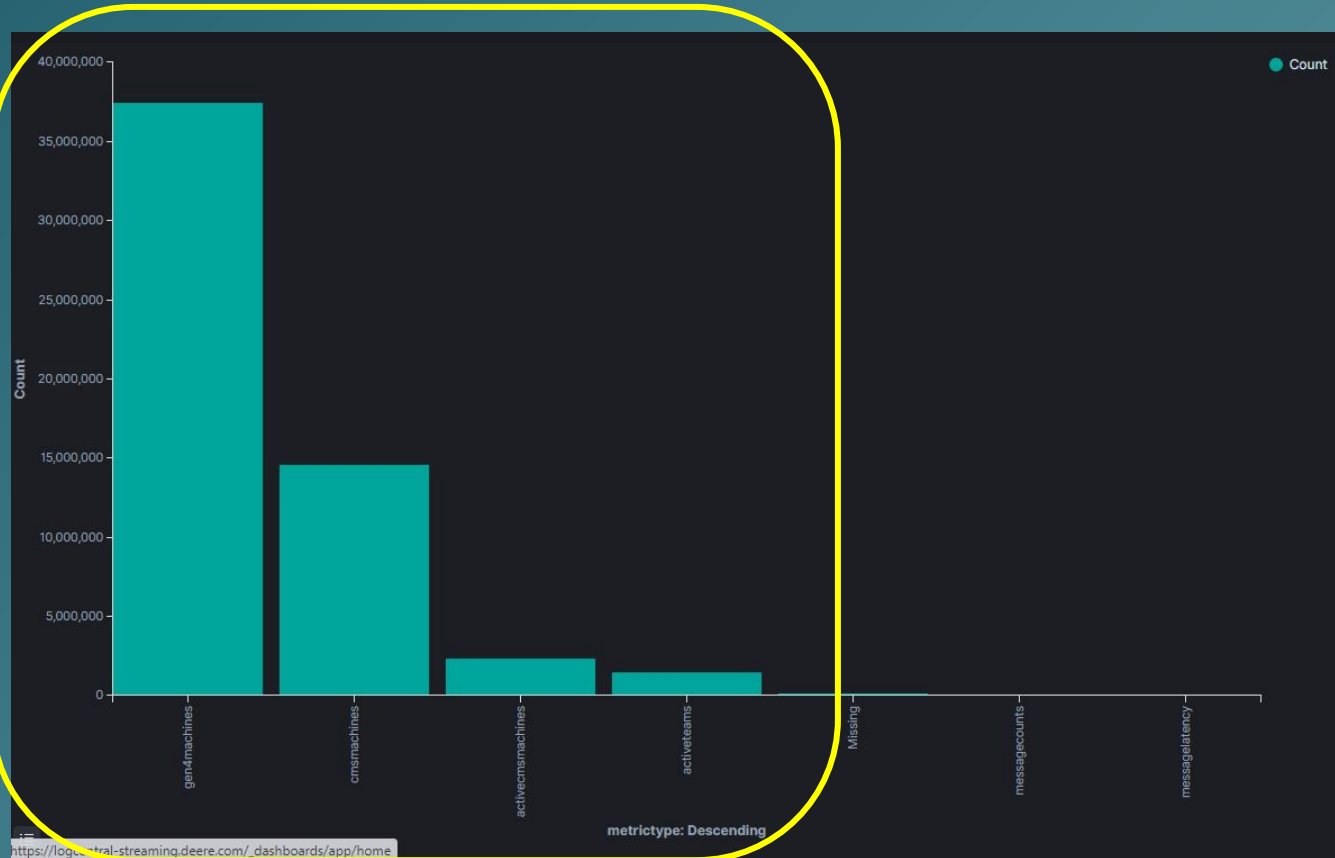
- Some of our key metrics became unresponsive as we grew
 - These involved counting distinct machines per minute
 - A 'cardinality' metric
- Let's have a look...
 - 5 min IFDS overview (domain doc)
 - Teams + machines graph
 - Metrics entries, data volumes chart, monthly storage

We count connected machines/time



This helps us spot problems and is an indicator of system usage and the number of customers which might be affected by any problem.

99% of our metrics are individual values



The four big buckets are mostly individual machine IDs logged by each instance of the app, e.g. 'machines observed per minute'



99% of our metrics are individual values

Time ▾	InstanceId	message	metrictype
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	eb3b4dfd-97e8-4139-b19f-c4e10493fd40	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	b20d89a1-a699-44c3-8278-b877a7d686f0	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	9bbdff80-1133-4fd3-bcdf-e104ea701b56	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	8fea86e4-47e6-4c05-91dc-c219ef41b652	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	2ac00870-6643-46a6-a8bd-0620896cd158	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	23ebd172-eb80-4e7b-9047-15b73a511843	activeteams
> Apr 12, 2022 @ 12:58:18.353	66554e93a14849078cc3e4c953700d0a	e1b29f76-50d9-4a70-9f9b-705faeb2d094	activeteams

Here we see individual values (team ids) logged - to be counted later

1% of our metrics are traditional

Time ▾	InstanceId	operationType	operationName	averageTimeInMs	maxTimeInMs
> Apr 12, 2022 @ 13:09:17.924	66554e93a14849078cc3e4c953700d0a	S3Call	upload	68	1,071
> Apr 12, 2022 @ 13:09:17.924	66554e93a14849078cc3e4c953700d0a	RedisCall	publish	3	401
> Apr 12, 2022 @ 13:09:17.923	66554e93a14849078cc3e4c953700d0a	S3Call	listObjectsV2	68	1,008
> Apr 12, 2022 @ 13:09:17.923	66554e93a14849078cc3e4c953700d0a	S3Call	getObject	347	1,422
> Apr 12, 2022 @ 13:09:10.314	d4d82a5475b14de59d8091b41cd61497	RedisCall	publish	2	485
> Apr 12, 2022 @ 13:09:10.314	d4d82a5475b14de59d8091b41cd61497	S3Call	upload	65	2,426
> Apr 12, 2022 @ 13:09:10.314	d4d82a5475b14de59d8091b41cd61497	S3Call	listObjectsV2	60	753

These metrics rows are logged in small numbers, once per minute per instance. They show message latency statistics, etc.



We have two kinds of metrics

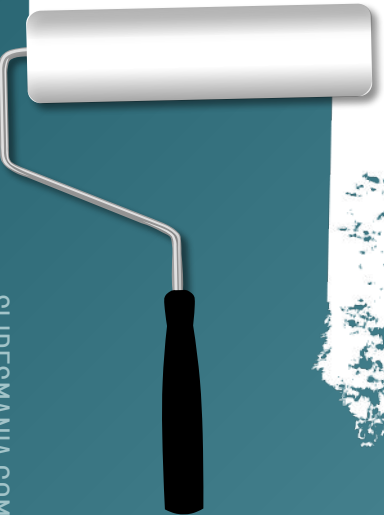
Aggregatable metrics

- 1% of the storage
- count, sum, min, max, p99
- aggregation works 'normally'
 - can count/max/sum across instances and intervals
- require very little storage

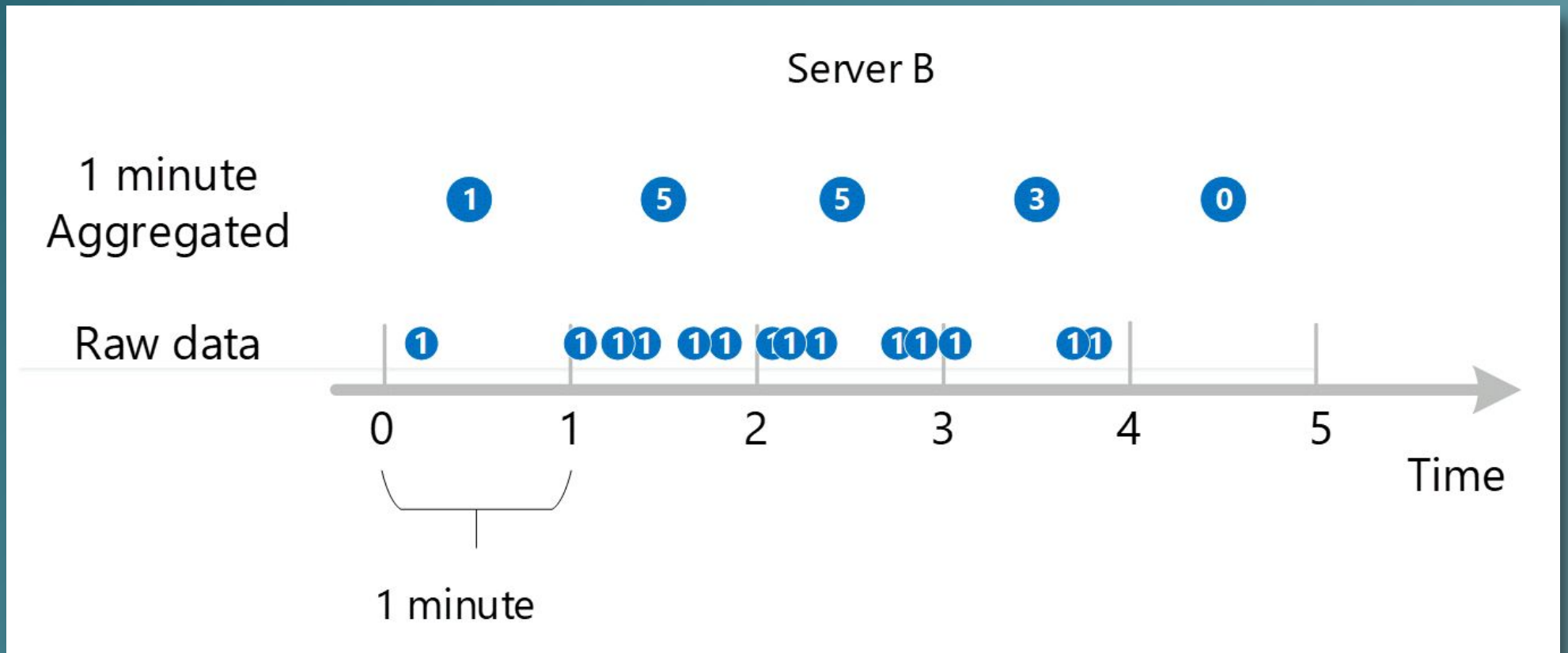
Non-aggregatable metrics

- 99% of the storage (> 500 GB/mo)
- count distinct values, etc.
- *aggregation of instance-level counts seems impossible!*
 - 'double counting' errors
- First solution: store all values and compute on-the-fly
- Too big/slow for real-time metrics on very large streams!

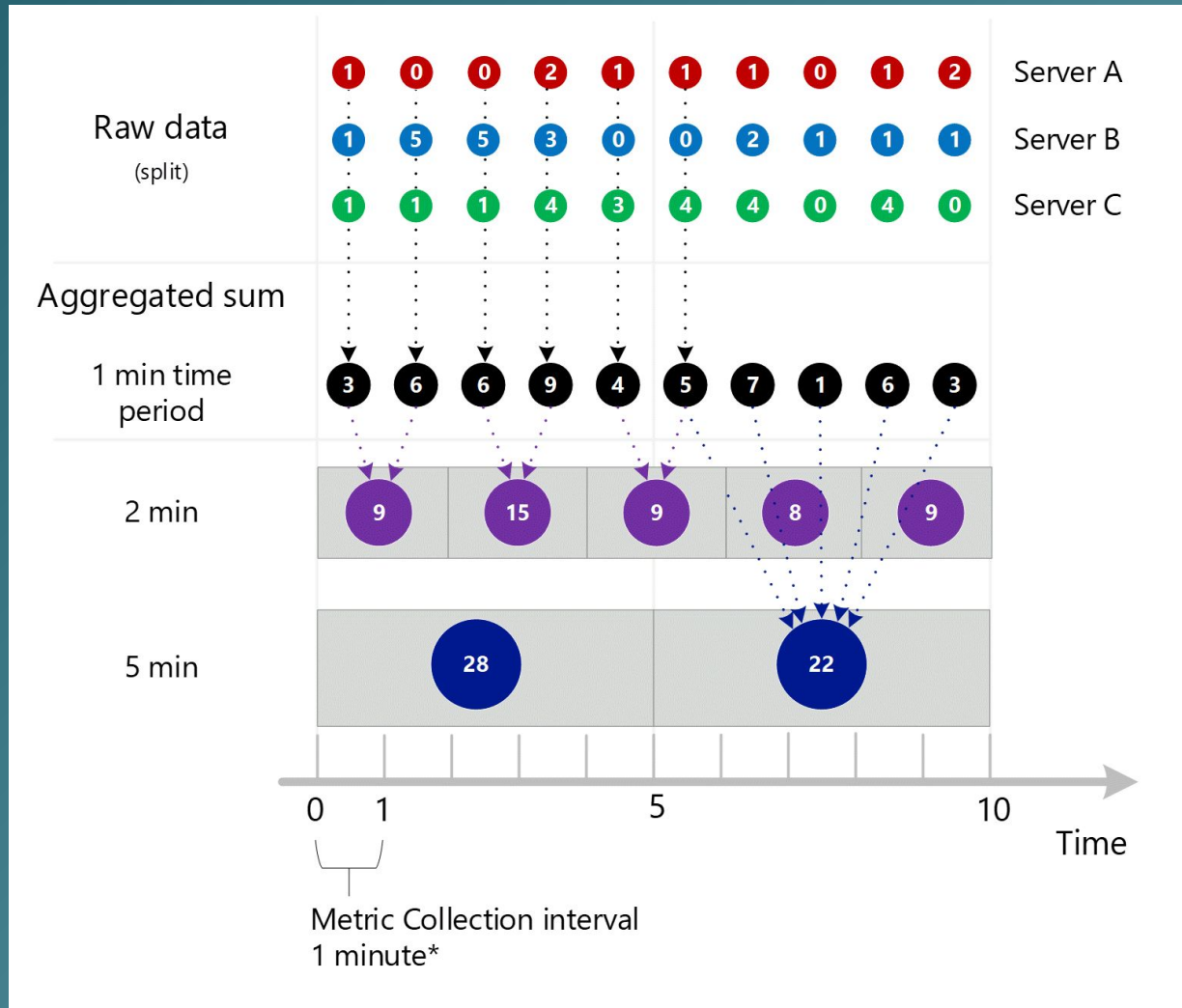
Why are some metrics aggregatable and others not?



Aggregating counts within an instance



Aggregation during queries



← In metrics db
(ElasticSearch for
us)

← ElasticSearch
queries aggregate
across instances
and minutes

What is the difference?

Aggregatable metrics all must:

- fundamentally behave like addition:
 - They combine two values of the same type
 - The result is the same type
 - `sum(a: Sum, b: Sum): Sum`
- If this were not the case, how would you write this code?

```
// Sum must return the same type as its args,  
// since total is used in both places  
[1,2,3,4,5].foreach {i =>  
  total = sum(total,i)  
}
```


What is the difference?

Aggregatable metrics all must:

- have a value that 'acts like zero', e.g. $a + 0 = a$

```
total = 0
[1,2,3,4,5].foreach {i =>
  total = sum(total,i)
}
```

```
// When maxing, -MAXINT is like zero
max = -MAXINT
[1,2,3,4,5].foreach {i =>
  max = max(result,i)
}
```

What is the difference?

Aggregatable metrics all must:

- not care how we partition, if we break the problem up

```
// given [1,2,3]  
sum(sum(1,2),3)
```

```
// is the same as  
sum(1,sum(2,3))
```

- this is critical for distribution
 - we don't always partition the same way

What is the difference?

Aggregatable metrics all must:

- not care about the order of arguments

```
// given [1,2,3]  
sum(sum(1,2),3)
```

```
// is the same as  
sum(3,sum(1,2))
```

- this is critical for parallelization
 - we don't know which instances report first

What is the difference?

Aggregatable metrics all must:

- fundamentally behave like addition
 - a *binary, closed function*
 - `sum(a: Sum, b: Sum): Sum`
 - `max(a: Max, b: Max): Max`
 - $a + (b + c) = (a + b) + c$
 - *associativity*
 - $a + 0 = a$ (value that 'acts like zero')
 - Identity element
 - This gives us a monoid
 - $a + b = b + a$
 - *commutativity*
 - This gives us a commutative monoid

There doesn't seem to be a function to aggregate 'distinct value counts'

//how could this work?

```
add(a: DistinctCount, b: DistinctCount): DistinctCount
```

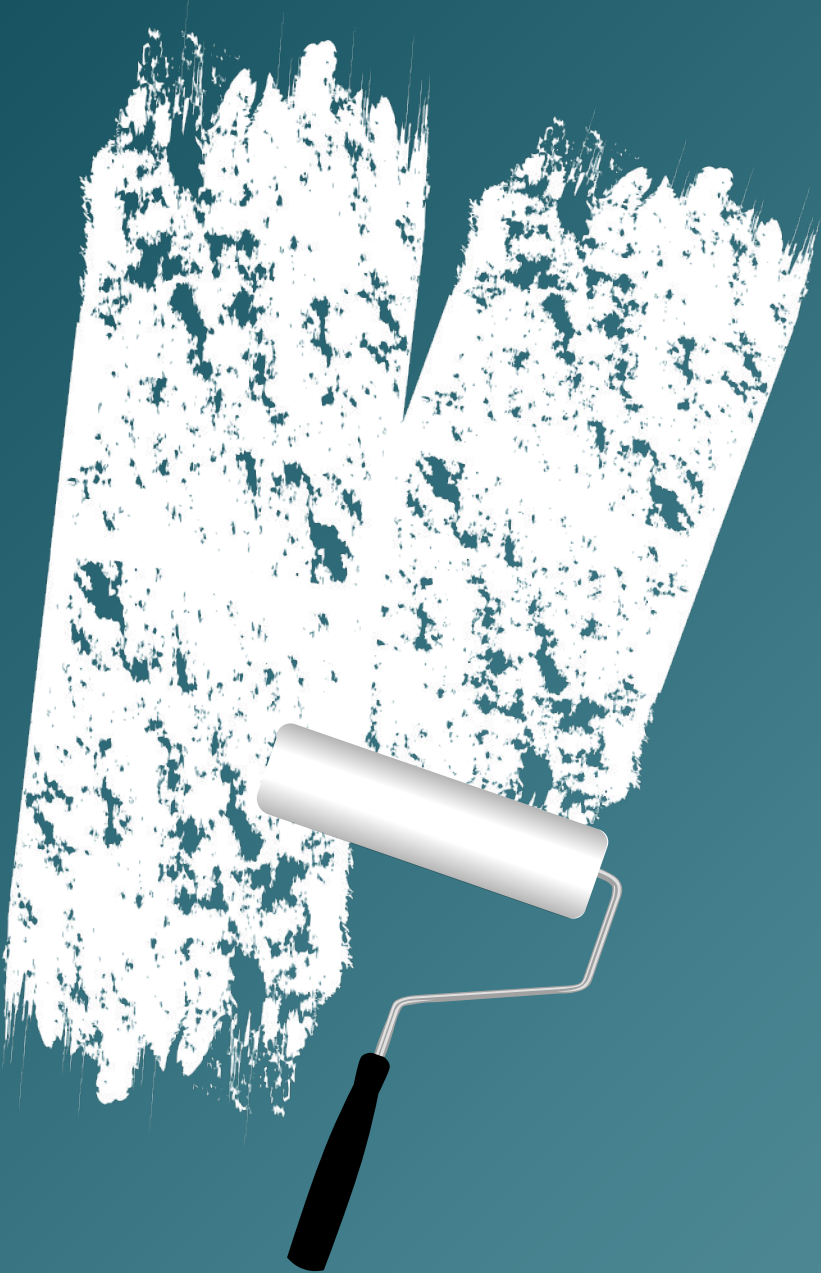


HyperLogLog is a monoid

- a probabilistic data structure + algorithm
- counts *approximate* distinct items
- From wikipedia:

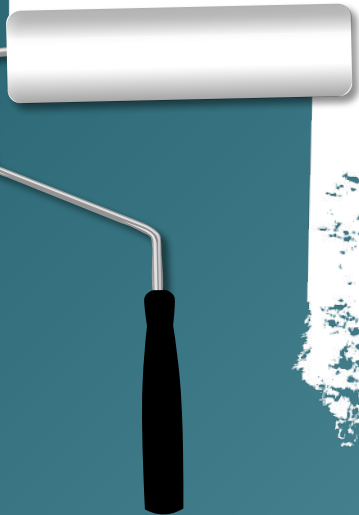
[...] able to estimate cardinalities of $> 10^9$ with a typical [...] std. error of 2%, using 1.5 kB of memory

- Supported in Postgres, Redis, AWS Redshift, Databricks, BigQuery, Druid, ElasticSearch (via plugin), Presto, etc.



How can we count all the things without counting all the things?

— Everybody when introduced to HyperLogLog



How HLL works

Let's dive in

- HLL has three main operations
 - `add(hashValue:Long) // O(1)`
 - `count():Long //O(registers)`
 - `union(a:HLL,b:HLL):HLL //O(registers)`
 - *This is the monoid '+' function*
- Storage grows as $O(\log(\log(\text{values})))$
 - very, very slowly

How HLL works

Let's dive in

- An HLL consists of a number of 'buckets'
 - tuneable for accuracy
- buckets represent subsets of all values seen
- incoming values are hashed to binary numbers
- values 'allocated' to a bucket based on first few bits
- each bucket holds the max number of leading zeroes observed in its values

How HLL works

Adding values

Using first 3 bits to determine bucket, there will be eight buckets (2^3).

Each bucket stores the maximum leading zeroes seen (MLZ)

Value	Hashed/Binary
Foo	→ 0010111001011 → bucket 1, MLZ = 1
Bar	→ 0110010101110 → bucket 3, MLZ = 2
Baz	→ 000001010101 → bucket 0, MLZ = 3
Buzz	→ 0110000111010 → bucket 3, MLZ = 4

Bucket:	0 1 2 3 4 5 6 7 8
MLZ:	3 1 0 4 0 0 0 0 0

How HLL works

Counting values

Counting values 'averages' the buckets:

Bucket:	0 1 2 3 4 5 6 7 8
MLZ:	3 1 0 4 0 0 0 0 0

Average (HM) = $3 / (1/3 + 1/1 + 1/4) = 1.89$

WTH? 1.89 != 4 :)

When few values have been seen, this isn't accurate. Libraries use some other tricks to improve accuracy (see java-hll 'promotion').

Accuracy for us has been ~98%!

How HLL works

Merging HLLs

Combining two HLLs is simply taking the max of their bucket values:

Bucket:	0 1 2 3 4 5 6 7 8	HLL 1
MLZ:	3 1 0 4 0 0 0 0 0	
Bucket:	0 1 2 3 4 5 6 7 8	HLL 2
MLZ:	2 4 1 1 0 2 0 3 0	
Bucket:	0 1 2 3 4 5 6 7 8	Result
MLZ:	3 4 1 4 0 2 0 3 0	

This is very fast! It also avoids double-counting (try it!)

How we added HLL metrics to IFDS

We didn't have to implement HLL

- We evaluated a few options and settled on Postgres' HLL support + the java-hll library
- Reasons for this included AWS support, team familiarity and the common storage spec used in these two.
 - Also see js-hll and python-hll



HLLs in memory:

```
// Empty HLL with tuning params
val gen4MachinesHLL = new HLL(11, 5)

// Add values as messages come in
val hashed = hash(displaySerialNumber)
gen4MachinesHLL.addRow(hashed)
```

- This is done every time a message comes in
- `new HLL(...)` is the monoid 'zero' that we need
- There is a 'merge' function for combining two HLLs
 - This is the monoid '+' function in java



Publishing to the metrics db:

```
// UPSERT Every minute
insert into gen4_machines (minute_bucket, bucket_hll)
  values (date_trunc('minute', $bucketTime), $bucketHLL)
on conflict (minute_bucket) DO UPDATE
  SET bucket_hll = hll_union(gen4_machines.bucket_hll,
    EXCLUDED.bucket_hll);
```

- There is a row (bucket) for each minute
- We truncate the timestamp to get the minute bucket
- **hll_union** function is the monoid '+' in SQL
- Values from different instances are aggregated into the minute buckets



Looking in the db:

```
// Inside 'gen4_machines' metric table  
select * from gen4_machines limit 3;
```

```
minute_bucket, bucket_hll  
2021-12-21 14:02, \x148b7f088440002000001080000006008[...]  
2021-12-21 14:04, \x148b7f1104420824304241a0211806208[...]  
2021-12-21 14:03, \x148b7f110442048000c411a0240844208[...]
```

- The HLL is a serialized form of the java hll
 - This works because there is a shared storage specification
- The HLLs here are about 2.6K each

How we added HLL metrics to IFDS

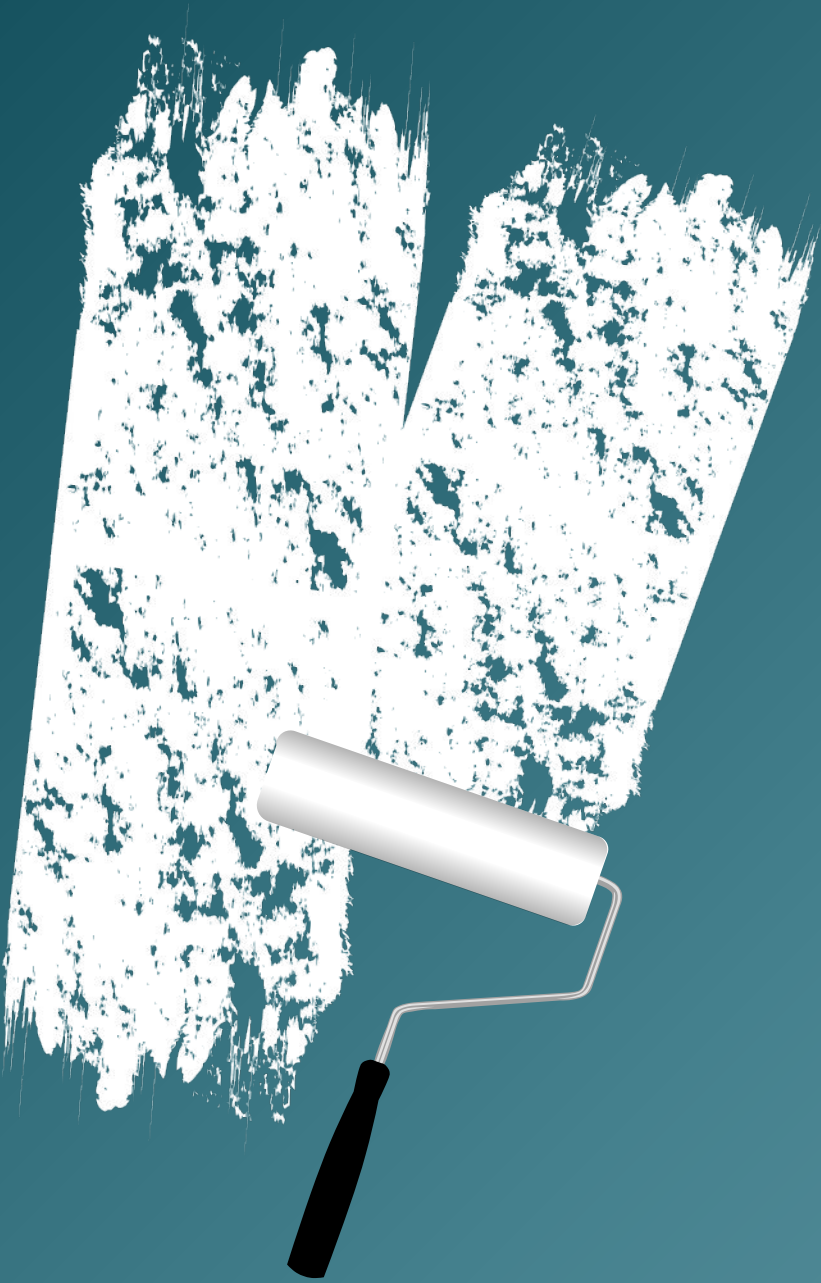
Changes we made

- We verified the hll extension is enabled in AWS Aurora Postgres
- We created small tables to hold HLLs per minute for each metric
- We changed the metrics collector to add values to the in-memory HLL
- We changed the metrics publisher to add its HLL to the Postgres record for that minute
- We added a grafana panel to aggregate HLL values

How we added HLL metrics to IFDS

Results

- We started using the HLL metrics on 12/21/21
- On 4/7/22
 - ~ 150K rows in each metric table
 - 496 MB (47 MB per month, per metric)
 - Elastic search used ~1400 GB (est.)
 - 99.9% reduction
- HLL panel in grafana is *much* faster
- Accuracy seems around 98%



Now that you know HLL, meet the family

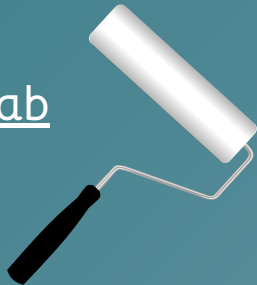
- See [Shawn's gist](#)
- bloom filter: *is X a member of the set?*
- count-min-sketch: *how frequent is value X ?*
- exponential histogram: *how many times did we see value X in n -minute windows?*
- min hasher: *how similar are two sets?*
- q tree: *histogram of values*

An incredible talk



Avi Bryant from stripe.com

<https://www.infoq.com/presentations/abstract-algebra-analytics/>





Resources

- <https://github.deere.com/v8znx3a/hyperloglog>
 - Has detailed instructions on how we experimented, data sets, timing results, etc.
- Links to many other resources are found in the above repo.



Thank you!